

Osnovi računarskih sistema

— prateći materijal za vežbe —

Milena Vujošević–Janičić i Jelena Tomašević

Sadržaj

1	Lavirint	7
1.1	Zadatak	7
1.2	Rešenje	8
2	Izrazi	19
2.1	Zadatak	19
2.2	Klasa Okolina	20
2.3	Klasa Izraz	22
2.4	Klasa Konstanta	22
2.5	Klasa Promenljiva	23
2.6	Klasa Zbir — osnovni elementi	23
2.7	Klasa Zbir	24
2.8	Metod Kopija()	25
2.9	Rešenje	27
3	Enciklopedija	35
4	Prostor imena	37
5	Tokovi	41
5.1	Datotečni tokovi	44
6	Izuzeci	51
7	Zadaci sa praktikuma	63
7.1	Matematički izrazi	63
7.2	Enciklopedija	66
7.3	Tokovi i izuzeci	66

Predgovor

Ovo je prateći materijal za vežbe koje držimo iz predmenta Osnovi računarskih sistema. On ne može zameniti pohađanje vežbi niti korišćenje druge preporučene literature. Većinu materijala čine zadaci i rešenja mr Saše Malkova (raspoloživi na <http://codd.matf.bg.ac.yu/ors/files2004smalkov/>) dok su naši prateći tekst, objašnjenja i neki primeri.

Zahvaljujemo svojim studentima na aktivnom učešću u nastavi čime su nam pomogli u uobličavanju ovog materijala.

Svi komentari i sugestije vezane za ovaj materijal biće veoma dobrodošli.

Milena Vujošević-Janičić

www.matf.bg.ac.yu/~milena

Jelena Tomašević

www.matf.bg.ac.yu/~jtomasevic

1

Lavirint

1.1 Zadatak

Lavirint je zapisan u datoteci "lavirint.dat" na sledeći način:

- u prvom redu zapisuju se praznim prostorom razdvojeni celi brojevi koji predstavljaju širinu i visinu lavirinta;
- zatim sledi onoliko redova kolika je visina lavirinta, a u svakom redu po onoliko znakova kolika je širina lavirinta;
- znak 'X' oznaka za zid, ' ' za put, 'O' za izlaz (ili cilj), a '@' za ulaz (ili početnu poziciju);
- mora postojati bar jedan ulaz i bar jedan izlaz.

Primer:

```
10 5
XXXXXXOXXX
X      X X
X XXXXXX X
X      X
XXX@XXXXXX
```

Napisati program koji čita lavirint, proverava da li je ispravno zapisan, pronalazi najkraći put od ulaza do izlaza i prikazuje na standardnom izlazu najkraći put ucrtan u lavirint upotrebom znaka '.', kao u narednom primeru:

```
10 5
XXXXXXOXXX
X.....X X
X.XXXXXX X
X.... X
XXX@XXXXXX
```

1.2 Rešenje

Na osnovu teksta zadatka, jasno je da je potrebno napraviti klasu Lavirint i u njoj obezbediti metod za pronalaženje najkraćeg puta. Takođe, potrebno je i predefinisati operatore << i >> kako bi omogućili jednostavno učitavanje i ispisivanje lavirinta. Na osnovu ovih zaključaka moguće je napisati main funkciju:

```
main()
{
    Lavirint l;
    ifstream f("lavirint.dat");
    f >> l;
    l.PronalazenjeNajkracegPut();
    cout << l;
    return 0;
}
```

Operatore << i >> implementiraćemo na uobičajni način, koristeći metode Citaj i Pisi.

```
istream& operator >> ( istream& istr, Lavirint& l )
{
    l.Citaj(istr);
    return istr;
}

ostream& operator << ( ostream& ostr, const Lavirint& l )
{
    l.Pisi(ostr);
    return ostr;
}
```

Sada je neophodno da razmišljamo o strukturi naše klase Lavirint i o tome šta sve ona treba da sadrži. Dakle, neophodno je naći ideju kako formirati najkraći put u lavirintu.

Prvo, jasno je da nam je potrebna nekakva matrica u kojoj ćemo čuvati ulaz iz datoteke. Recimo da to može da bude niz nizova karaktera (rešenje u programskom jeziku C) odnosno vektor čiji su elementi vektori karaktera. Neka se ta matrica zove `_mapa`.

Recimo da smo pronašli najkraći put, gde će se on čuvati? Taj put možemo da smestimo u matricu čiji su elementi tipa bool i za svaki element važi da je true ako pripada putu odnosno da je false ako putu ne pripada.

Na osnovu ove dve matrice možemo da uradimo štampanje rezultata, ako je neki element matrice pripada putu onda se štampa tačka, ako ne pripada onda se štampa ono što je inače u mapi.

Potrebno je čuvati i koordinate početne pozicije u lavirintu kao i podatak da li je put pronađen (ne mora svaki lavirint da ima put do izlaza).

Šta dalje?

Pretpostavimo da je nula početna pozicija i da nemamo zidova. Posmatrajmo sledeću sliku i kretanje iz svake tačke na sve četiri strane bez vraćanja. Na primer, do polja koja su obeležena brojem tri, nije moguće stići za manje od tri koraka, ali je naravno moguće stići za više od tri koraka.

```

      3
    3 2 3
  3 2 1 2 3
3 2 1 0 1 2 3
  3 2 1 2 3
    3 2 3
      3

```

Da bismo obezbedili kretanje bez vraćanja potrebno je negde da čuvamo podatke o poljima na kojima smo već bili. Za to nam je potrebna i treća matrica.

Pošto su nam potrebne tri matrice, nameće se da bi bilo dobro napraviti šablon klase matrica i onda upotrebljavati taj šablon kako bismo imali udobniji rad i pristup članovima matrice.

```

template <class T>
class Matrica
{
public:
    Matrica()
        {}

    Matrica( unsigned s, unsigned v, T t )
        : _podaci(s)
        {
            // proveru da su s i v > 0 ostavljamo za drugu priliku
            for( unsigned i=0; i<s; i++ ){
                // _podaci[i] = vector<T>(v);
                _podaci[i].resize(v);
                for( unsigned j=0; j<v; j++ )
                    _podaci[i][j] = t;
            }
        }

    unsigned Sirina() const
        { return _podaci.size(); }

    unsigned Visina() const
        { return _podaci[0].size(); }

```

```

vector<T>& operator[]( unsigned i )
    { return _podaci[i]; }

const vector<T>& operator[]( unsigned i ) const
    { return _podaci[i]; }

private:
    vector< vector<T> > _podaci;
};

```

Sada znamo neke privatne podatke klase Lavirint

```

private:
    Matrica<char> _mapa;
    Matrica<bool> _obradjeno;
    Matrica<bool> _put;
    unsigned xPocetka, yPocetka;
    bool pronadjenPut;

i možemo da napišemo metode Pisi i Citaj.

void Citaj( istream& istr )
{
    unsigned s,v;
    istr >> s >> v;
    _mapa = Matrica<char>( s, v, ' ' );
    for( unsigned i=0; i<v; i++ ){
        istr >> ws;
        for( unsigned j=0; j<s; j++ )
            _mapa[j][i] = istr.get();
    }
}

```

```

void Pisi( ostream& ostr ) const
{
    unsigned
        s = _mapa.Sirina(),
        v = _mapa.Visina();
    ostr << s << ' ' << v << endl;
    for( unsigned i=0; i<v; i++ ){
        for( unsigned j=0; j<s; j++ )
            if( pronadjenPut && _put[j][i] )
                ostr << '.';
            else
                ostr << _mapa[j][i];
    }
}

```

```

        ostr << endl;
    }
}

```

Takođe, možemo da implementiramo konstruktor bez argumenata klase Lavirint.

```

Lavirint()
: pronadjenPut(false)
{}

```

Do sada smo "sve" uradili osim implemetacije algoritma pronalaženja najkraćeg puta. Držaćemo se ideje odlaska na sve četiri strane onda kada je to moguće. Kada stignemo do izlaza pronašli smo put, ali u međuvremenu moramo nekako da čuvamo kuda smo išli i odakle smo došli. Te podatke stavljaćemo u jedan niz čiji će argumenti biti koordinata tačke u koju smo došli i indeks tačke u nizu iz koje smo došli. Ta tri podatka apstrahovaćemo u klasu Pozicija koju ćemo definisati u okviru klase Lavirint.

```

class Pozicija {
public:
    unsigned x,y;
    int prethodna;
    Pozicija( unsigned _x, unsigned _y, int _p )
        : x(_x), y(_y), prethodna(_p)
    {}
};

```

Dakle, u okviru klase Lavirint pamtimo još jedan podatak koji će da čuva putanju.

```
vector<Pozicija> putanja;
```

U tu putanju, prvo ubacujemo poziciju početne tačke. Koja je to pozicija? Moramo je pronaći na osnovu mape.

```

void PronalazenjePocetaka()
{
    unsigned
        s = _mapa.Sirina(),
        v = _mapa.Visina();
    for( unsigned i=0; i<v; i++ )
        for( unsigned j=0; j<s; j++ )
            if( _mapa[j][i] == '@' )
                putanja.push_back( Pozicija(j,i,-1) );
}

```

Pre nego što neko polje ubacimo u putanju moramo da proverimo da ono nije već obrađeno, da kroz njega može da se prođe ili da li je to polje koje označava izlaz.

```

void proveraPolja( unsigned x, unsigned y, int prethodno )
{
    if( _mapa[x][y] == '0' ){
        pronadjenPut = true;
    }
    else if( _mapa[x][y] == ' ' && !_obradjeno[x][y] ){
        putanja.push_back( Pozicija(x,y,prethodno) );
        _obradjeno[x][y] = true;
    }
}

```

Kako pronaći najkraći put? Treba implementirati ideju sa odlaskom na sve četiri strane tj na one koje je moguće otići. Put kojim idemo pamtimo u nizu putanja.

```

void PronalazenjeNajkracegPut()
{
    // pocinjemo trazenje
    putanja.clear();
    PronalazenjePocetaka();

    unsigned tekuca = 0;
    pronadjenPut = false;
    _obradjeno = Matrica<bool>( _mapa.Sirina(), _mapa.Visina(), false );

    // trazimo
    while( tekuca < putanja.size() && !pronadjenPut ){
        if( putanja[tekuca].y > 0 )
            proveraPolja( putanja[tekuca].x, putanja[tekuca].y-1,tekuca);
        if( putanja[tekuca].y < _mapa.Visina()-1 )
            proveraPolja( putanja[tekuca].x, putanja[tekuca].y+1,tekuca);
        if( putanja[tekuca].x > 0 )
            proveraPolja( putanja[tekuca].x-1, putanja[tekuca].y,tekuca);
        if( putanja[tekuca].x < _mapa.Sirina()-1 )
            proveraPolja( putanja[tekuca].x+1, putanja[tekuca].y,tekuca);
        tekuca++;
    }

    if( pronadjenPut )
        RestauracijaPutanje( tekuca-1 );
}

```

Na osnovu formiranja putanje moramo da rekonstruišemo najkraći put tj put kojim smo došli do izlaza.

```

void RestauracijaPutanje( int poslednje )

```

```
{
    _put = Matrica<bool>( _mapa.Sirina(), _mapa.Visina(), false );
    for( int i=poslednje; i>0; i=putanja[i].prethodna )
        _put[putanja[i].x][putanja[i].y] = true;
}
```

Kompletno rešenje:

```
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

template <class T>
class Matrica
{
public:
    Matrica()
        {}

    Matrica( unsigned s, unsigned v, T t )
        : _podaci(s)
        {
            // proveru da su s i v > 0 ostavljamo za drugu priliku
            for( unsigned i=0; i<s; i++ ){
                // _podaci[i] = vector<T>(v);
                _podaci[i].resize(v);
                for( unsigned j=0; j<v; j++ )
                    _podaci[i][j] = t;
            }
        }

    unsigned Sirina() const
        { return _podaci.size(); }

    unsigned Visina() const
        { return _podaci[0].size(); }

    vector<T>& operator[]( unsigned i )
        { return _podaci[i]; }

    const vector<T>& operator[]( unsigned i ) const
        { return _podaci[i]; }
}
```

```

private:
    vector< vector<T> > _podaci;
};

class Lavirint
{
public:
    class Pozicija {
    public:
        unsigned x,y;
        int prethodna;
        Pozicija( unsigned _x, unsigned _y, int _p )
            : x(_x), y(_y), prethodna(_p)
        {}
    };

    Lavirint()
        : pronadjenPut(false)
    {}

    void PronalazenjePocetaka()
    {
        unsigned
            s = _mapa.Sirina(),
            v = _mapa.Visina();
        for( unsigned i=0; i<v; i++ )
            for( unsigned j=0; j<s; j++ )
                if( _mapa[j][i] == '@' )
                    putanja.push_back( Pozicija(j,i,-1) );
    }

    void PronalazenjeNajkracegPut()
    {
        // pocinjemo trazenje
        putanja.clear();
        PronalazenjePocetaka();

        unsigned tekuca = 0;
        pronadjenPut = false;
        _obradjeno = Matrica<bool>( _mapa.Sirina(), _mapa.Visina(), false );

        // trazimo
        while( tekuca < putanja.size() && !pronadjenPut ){
            if( putanja[tekuca].y > 0 )

```

```

        proveraPolja( putanja[tekuca].x, putanja[tekuca].y-1,tekuca);
    if( putanja[tekuca].y < _mapa.Visina()-1 )
        proveraPolja( putanja[tekuca].x, putanja[tekuca].y+1,tekuca);
    if( putanja[tekuca].x > 0 )
        proveraPolja( putanja[tekuca].x-1, putanja[tekuca].y,tekuca);
    if( putanja[tekuca].x < _mapa.Sirina()-1 )
        proveraPolja( putanja[tekuca].x+1, putanja[tekuca].y,tekuca);
    tekuca++;
}

if( pronadjenPut )
    RestauracijaPutanje( tekuca-1 );
}

void RestauracijaPutanje( int poslednje )
{
    _put = Matrica<bool>( _mapa.Sirina(), _mapa.Visina(), false );
    for( int i=poslednje; i>0; i=putanja[i].prethodna )
        _put[putanja[i].x][putanja[i].y] = true;
}

void proveraPolja( unsigned x, unsigned y, int prethodno )
{
    if( _mapa[x][y] == '0' ){
        pronadjenPut = true;
    }
    else if( _mapa[x][y] == ' ' && !_obradjeno[x][y] ){
        putanja.push_back( Pozicija(x,y,prethodno) );
        _obradjeno[x][y] = true;
    }
}

void Citaj( istream& istr )
{
    unsigned s,v;
    istr >> s >> v;
    _mapa = Matrica<char>( s, v, ' ' );
    for( unsigned i=0; i<v; i++){
        istr >> ws;
        for( unsigned j=0; j<s; j++ )
            _mapa[j][i] = istr.get();
    }
}

```

```

void Pisi( ostream& ostr ) const
{
    unsigned
        s = _mapa.Sirina(),
        v = _mapa.Visina();
    ostr << s << ' ' << v << endl;
    for( unsigned i=0; i<v; i++ ){
        for( unsigned j=0; j<s; j++ )
            if( pronadjenPut && _put[j][i] )
                ostr << '.';
            else
                ostr << _mapa[j][i];
        ostr << endl;
    }
}

private:
    Matrica<char> _mapa;
    Matrica<bool> _obradjeno;
    Matrica<bool> _put;
    vector<Pozicija> putanja;
    unsigned xPocetka, yPocetka;
    bool pronadjenPut;
};

istream& operator >> ( istream& istr, Lavirint& l )
{
    l.Citaj(istr);
    return istr;
}

ostream& operator << ( ostream& ostr, const Lavirint& l )
{
    l.Pisi(ostr);
    return ostr;
}

main()
{
    Lavirint l;
    ifstream f("lavirint.dat");
    f >> l;
    l.PronalazenjeNajkracegPut();
    cout << l;
}

```



```
    return 0;  
}
```


2

Izrazi

2.1 Zadatak

1. Napisati klasu `Okolina` koja predstavlja skup promenljivih i njima dodeljenih realnih brojeva. Obezbediti metode
 - `void DodajPromenljivu(string naziv, double vrednost)` dodaje okolini novu promenljivu
 - `double VrednostPromenljive(const string& s)` vraća vrednost date promenljive ili proizvodi izuzetak ako ona nije definisana
 - `bool DefinisanaPromenljiva(const string& s)` proverava da li je definisana data promenljiva
 - `bool BrisanjePromenljive(const string& naziv)` uklanja promenljivu iz okoline i vraća `true` ako je ona postojala, a `false` inače
2. Napisati klasu `Izraz` za predstavljanje izraza. Jedan složeni izraz može imati proizvoljno mnogo promenljivih. Obezbediti metode:
 - `double vrednost(Okolina&)` — vraća vrednost izraza u datoj okolini (tj. za date vrednosti promenljivih), a ako nisu definisane vrednosti svih promenljivih proizvodi se izuzetak.
 - `Izraz* uprosti(Okolina&)` koji za rezultat ima izraz koji se dobija uprošćavanjem izraza u datoj okolini (npr izraz $(x + (y + 5))^2$ za $y=3$ posle uprošćavanja postaje $(x + 8)^2$)
 - metod za ispisivanje izraza
 - sve ostale neophodne metode
3. Napisati klasu `Promenljiva` koja predstavlja promenljivu koja učestvuje u izrazu. Obezbediti:
 - konstruktor sa datim imenom promenljive
 - metode za izračunavanje i uprošćavanje u datoj okolini
 - metod za ispisivanje

- sve ostale neophodne metode
4. Napisati klasu **Konstanta** koja predstavlja realnu konstantu koja učestvuje u izrazu. Obezbediti:
- konstruktor sa datom vrednošću
 - metode za izačunavanje i uprošćavanje u datoj okolini
 - metod za ispisivanje
 - sve ostale neophodne metode
5. Napisati klase **Zbir**, **Razlika**, **Proizvod** i **Kolicnik**, koje služe za formiranje složenijih izraza i predstavljaju redom sabiranje, oduzimanje, množenje i deljenje dva izraza. U svakoj od tih klasa obezbediti
- konstruktor sa datim operandima
 - metode za izačunavanje i uprošćavanje u datoj okolini
 - metod za ispisivanje
 - sve ostale neophodne metode

Operatorski izrazi se zapisuju u formatu: otvorena zagrada, ime operatora, prvi argument, blanko, drugi argument, zatvorena zagrada. Npr. izraz $(x - a_2) * (y + 7.23)$ se prikazuje ovako: (PUTA (MINUS x a2) (PLUS y 7.23))

2.2 Klasa Okolina

```
#include <string>
#include <map>
#include <iostream>

using namespace std;

class Okolina
{
public:
    double VrednostPromenljive( const string& s ) const
    {
        map<string,double>::const_iterator
            f = _Promenljive.find(s);
        if( f != _Promenljive.end() )
            return f->second;
        else{
            // ovde bi trebalo da se napravi izuzetak
            return 0;
        }
    }
}
```

```
bool DefinisanaPromenljiva( const string& s ) const
{
    map<string,double>::const_iterator
        f = _Promenljive.find(s);
    return f != _Promenljive.end();
}

void DodajPromenljivu( const string& s, double v )
{ _Promenljive[s] = v; }

bool BrisanjePromenljive(const string& naziv)
{
    map<string, double>::iterator f = _Promenljive.find(naziv);
    if( f != _Promenljive.end() ) {
        _Promenljive.erase(f);
        return true;
    }
    else
        return false;
}

private:
    map<string, double> _Promenljive;
};

Da bi smo isprobali kako funkcioniše ova klasa možemo koristiti sledeću main
funkciju.

main()
{
    Okolina o;
    o.DodajPromenljivu( "x", 2.32 );
    o.DodajPromenljivu( "y", 5 );
    o.DodajPromenljivu( "x", 7 );

    cout << o.VrednostPromenljive( "x" ) << endl;
    cout << ( o.DefinisanaPromenljiva("y") ? "Imamo y." : "Nemamo y." ) << endl;
    cout << ( o.DefinisanaPromenljiva("z") ? "Imamo z." : "Nemamo z." ) << endl;

    o.BrisanjePromenljive("y");
    cout << o.VrednostPromenljive( "x" ) << endl;
    cout << ( o.DefinisanaPromenljiva("y") ? "Imamo y." : "Nemamo y." ) << endl;
    cout << ( o.DefinisanaPromenljiva("z") ? "Imamo z." : "Nemamo z." ) << endl;

    return 0;
}
```

```
}  
    /*  
    7  
    Imamo y.  
    Nemamo z.  
    7  
    Nemamo y.  
    Nemamo z.  
    */
```

2.3 Klasa Izraz

```
class Izraz  
{  
public:  
    virtual ~Izraz()  
        {}  
    virtual double Vrednost( const Okolina& o ) const = 0;  
    virtual void Ispisi( ostream& ostr ) const = 0;  
};  
  
ostream& operator << ( ostream& ostr, const Izraz& i )  
{  
    i.Ispisi( ostr );  
    return ostr;  
}
```

2.4 Klasa Konstanta

```
class Konstanta : public Izraz  
{  
public:  
    Konstanta( double d )  
        : _Vrednost(d)  
        {}  
  
    double Vrednost( const Okolina& o ) const  
        { return _Vrednost; }  
  
    void Ispisi( ostream& ostr ) const  
        { ostr << _Vrednost; }  
  
private:  
    double _Vrednost;
```

```
};
```

2.5 Klasa Promenljiva

```
class Promenljiva : public Izraz
{
public:
    Promenljiva( const string& s )
        : _Naziv(s)
        {}

    double Vrednost( const Okolina& o ) const
        { return o.VrednostPromenljive( _Naziv ); }

    void Ispisi( ostream& ostr ) const
        { ostr << _Naziv; }

private:
    string _Naziv;
};
```

2.6 Klasa Zbir — osnovni elementi

```
class Zbir : public Izraz
{
public:
    Zbir( Izraz* i1, Izraz* i2 )
        {}

    double Vrednost( const Okolina& o ) const
        { return 0; }

    void Ispisi( ostream& ostr ) const
        {}
};
```

Program koji koristi prethodno definisane klase:

```
main()
{
    Okolina o;
    o.DodajPromenljivu( "x", 5 );

    // izraz (x+2)
    // Izraz* i1 = new Zbir( new Promenljiva("x"), new Konstanta(2));
```

```

    Izraz* i1 = new Promenljiva("x");
    cout << (*i1) << " = " << i1->Vrednost( o ) << endl;
    delete i1;

    return 0;
}

```

2.7 Klasa Zbir

```

class Zbir : public Izraz
{
public:
    Zbir( Izraz* i1, Izraz* i2 )
        : _I1(i1), _I2(i2)
        {}

    ~Zbir()
    {
        delete _I1;
        delete _I2;
    }

    Zbir( const Zbir& z )
        {}

    Zbir& operator = ( const Zbir& z )
        {}

    double Vrednost( const Okolina& o ) const
    { return _I1->Vrednost(o) + _I2->Vrednost(o); }

    void Ispisi( ostream& ostr ) const
    { ostr << "( PLUS " << *_I1 << ' ' << *_I2 << " )"; }

private:
    Izraz *_I1, *_I2;
};

main()
{
    Okolina o;
    o.DodajPromenljivu( "x", 5 );

    // izraz (x+2)

```



```

    Izraz* i1 = new Zbir( new Promenljiva("x"), new Konstanta(2));
    cout << (*i1) << " = " << i1->Vrednost( o ) << endl;
    delete i1;

    return 0;
}

```

2.8 Metod Kopija()

```

class Izraz
{
public:
    virtual ~Izraz()
        {}
    virtual double Vrednost( const Okolina& o ) const = 0;
    virtual void Ispisi( ostream& ostr ) const = 0;
    virtual Izraz* Kopija() const = 0;
};

ostream& operator << ( ostream& ostr, const Izraz& i )
{
    i.Ispisi( ostr );
    return ostr;
}

class Konstanta : public Izraz
{
public:
    Konstanta( double d )
        : _Vrednost(d)
        {}

    double Vrednost( const Okolina& o ) const
        { return _Vrednost; }

    void Ispisi( ostream& ostr ) const
        { ostr << _Vrednost; }

    Konstanta* Kopija() const
        { return new Konstanta(*this); }

private:
    double _Vrednost;
};

```

```
class Promenljiva : public Izraz
{
public:
    Promenljiva( const string& s )
        : _Naziv(s)
        {}

    double Vrednost( const Okolina& o ) const
        { return o.VrednostPromenljive( _Naziv ); }

    void Ispisi( ostream& ostr ) const
        { ostr << _Naziv; }

    Promenljiva* Kopija() const
        { return new Promenljiva(*this); }

private:
    string _Naziv;
};

class Zbir : public Izraz
{
public:
    Zbir( Izraz* i1, Izraz* i2 )
        : _I1(i1), _I2(i2)
        {}

    ~Zbir()
    {
        delete _I1;
        delete _I2;
    }

    Zbir( const Zbir& z )
        : _I1( z._I1->Kopija() ),
          _I2( z._I2->Kopija() )
        {}

    Zbir& operator = ( const Zbir& z )
    {
        if( this != &z ){
            delete _I1;
            delete _I2;
```

```

        _I1 = z._I1->Kopija();
        _I2 = z._I2->Kopija();
    }
    return *this;
}

double Vrednost( const Okolina& o ) const
{ return _I1->Vrednost(o) + _I2->Vrednost(o); }

void Ispisi( ostream& ostr ) const
{ ostr << "( PLUS " << *_I1 << ' ' << *_I2 << " )"; }

Zbir* Kopija() const
{ return new Zbir(*this); }

private:
    Izraz *_I1, *_I2;
};

main()
{
    Okolina o;
    o.DodajPromenljivu( "x", 5 );

    // izraz (x+2)
    Zbir* z1 = new Zbir( new Promenljiva("x"), new Konstanta(2));
    Izraz* i1 = new Zbir(*z1);
    cout << (*i1) << " = " << i1->Vrednost( o ) << endl;
    delete i1;
    delete z1;

    return 0;
}

```

2.9 Rešenje

```

#include <string>
#include <map>
#include <iostream>

using namespace std;

class Okolina
{

```

```
public:
    double VrednostPromenljive( const string& s ) const
    {
        map<string,double>::const_iterator
            f = _Promenljive.find(s);
        if( f != _Promenljive.end() )
            return f->second;
        else{
            // ovde bi trebalo da se napravi izuzetak
            return 0;
        }
    }

    bool DefinisanaPromenljiva( const string& s ) const
    {
        map<string,double>::const_iterator
            f = _Promenljive.find(s);
        return f != _Promenljive.end();
    }

    void DodajPromenljivu( const string& s, double v )
    { _Promenljive[s] = v; }

    bool BrisanjePromenljive(const string& naziv)
    {
        map<string, double>::iterator f = _Promenljive.find(naziv);
        if( f != _Promenljive.end() ) {
            _Promenljive.erase(f);
            return true;
        }
        else
            return false;
    }

private:
    map<string,double> _Promenljive;
};

class Izraz
{
public:
    virtual ~Izraz()
    {}
};
```

```
virtual double Vrednost( const Okolina& o ) const = 0;
virtual void Ispisi( ostream& ostr ) const = 0;
virtual Izraz* Kopija() const = 0;
virtual Izraz* Uprosti( const Okolina& o ) const = 0;
virtual bool JesteKonstanta() const
    { return false; }
};

ostream& operator << ( ostream& ostr, const Izraz& i ) {
    i.Ispisi( ostr );
    return ostr;
}

class Konstanta : public Izraz
{
public:
    Konstanta( double d )
        : _Vrednost(d)
    {}

    double Vrednost( const Okolina& o ) const
        { return _Vrednost; }

    void Ispisi( ostream& ostr ) const
        { ostr << _Vrednost; }

    Konstanta* Kopija() const
        { return new Konstanta(*this); }

    Izraz* Uprosti( const Okolina& o ) const
        { return Kopija(); }

    bool JesteKonstanta() const
        { return true; }

private:
    double _Vrednost;
};

class Promenljiva : public Izraz
{
public:
    Promenljiva( const string& s )
        : _Naziv(s)
```

```
    {}

    double Vrednost( const Okolina& o ) const
    { return o.VrednostPromenljive( _Naziv ); }

    void Ispisi( ostream& ostr ) const
    { ostr << _Naziv; }

    Promenljiva* Kopija() const
    { return new Promenljiva(*this); }

    Izraz* Uprosti( const Okolina& o ) const
    {
        if( o.DefinisanaPromenljiva(_Naziv) )
            return new Konstanta( o.VrednostPromenljive(_Naziv));
        else
            return Kopija();
    }

private:
    string _Naziv;
};

class BinarniOperator : public Izraz
{
public:
    BinarniOperator( Izraz* i1, Izraz* i2 )
        : _I1(i1), _I2(i2)
    {}

    ~BinarniOperator()
    {
        delete _I1;
        delete _I2;
    }

    BinarniOperator( const BinarniOperator& z )
        : _I1( z._I1->Kopija() ),
          _I2( z._I2->Kopija() )
    {}

    BinarniOperator& operator = ( const BinarniOperator& z )
    {
        if( this != &z ){
```

```

        delete _I1;
        delete _I2;
        _I1 = z._I1->Kopija();
        _I2 = z._I2->Kopija();
    }
    return *this;
}

double Vrednost( const Okolina& o ) const
{ return Izracunaj( _I1->Vrednost(o), _I2->Vrednost(o)); }

Izraz* Uprosti( const Okolina& o ) const
{
    Izraz* i1 = _I1->Uprosti(o);
    Izraz* i2 = _I2->Uprosti(o);
    if( i1->JesteKonstanta() && i2->JesteKonstanta() ){
        Izraz* r = new Konstanta( Izracunaj( i1->Vrednost(o), i2->Vrednost(o)) );
        delete i1;
        delete i2;
        return r;
    }
    else
        return NapraviNovi( i1, i2 );
}

void Ispisi( ostream& ostr ) const
{ ostr << "( " << Naziv() << ' ' << *_I1 << ' ' << *_I2 << " )"; }

protected:
    virtual double Izracunaj( double x, double y ) const = 0;
    virtual Izraz* NapraviNovi( Izraz* i1, Izraz* i2 ) const = 0;
    virtual string Naziv() const = 0;

private:
    Izraz *_I1, *_I2;
};

class Zbir : public BinarniOperator
{
public:
    Zbir( Izraz* i1, Izraz* i2 )
        : BinarniOperator( i1, i2 )
    {}
};

```

```
Zbir* Kopija() const
    { return new Zbir(*this); }

protected:
    double Izracunaj( double x, double y ) const
        { return x + y; }

    Izraz* NapraviNovi( Izraz* i1, Izraz* i2 ) const
        { return new Zbir( i1, i2 ); }

    string Naziv() const
        { return "PLUS"; }
};

class Razlika : public BinarniOperator
{
public:
    Razlika( Izraz* i1, Izraz* i2 )
        : BinarniOperator( i1, i2 )
        {}

    Razlika* Kopija() const
        { return new Razlika(*this); }

protected:
    double Izracunaj( double x, double y ) const
        { return x - y; }

    Izraz* NapraviNovi( Izraz* i1, Izraz* i2 ) const
        { return new Razlika( i1, i2 ); }

    string Naziv() const
        { return "MINUS"; }
};

class Proizvod : public BinarniOperator
{
public:
    Proizvod( Izraz* i1, Izraz* i2 )
        : BinarniOperator( i1, i2 )
        {}

    Proizvod* Kopija() const
        { return new Proizvod(*this); }
```



```

protected:
    double Izracunaj( double x, double y ) const
        { return x * y; }

    Izraz* NapraviNovi( Izraz* i1, Izraz* i2 ) const
        { return new Proizvod( i1, i2 ); }

    string Naziv() const
        { return "PUTA"; }
};

class Kolicnik : public BinarniOperator
{
public:
    Kolicnik( Izraz* i1, Izraz* i2 )
        : BinarniOperator( i1, i2 )
        {}

    Kolicnik* Kopija() const
        { return new Kolicnik(*this); }

protected:
    double Izracunaj( double x, double y ) const
        { return x / y; }

    Izraz* NapraviNovi( Izraz* i1, Izraz* i2 ) const
        { return new Kolicnik( i1, i2 ); }

    string Naziv() const
        { return "PODELJENO"; }
};

main()
{
    Okolina o;
    o.DodajPromenljivu( "x", 5 );

    // izraz (x/2)+(y-(3.14*z))
    Izraz* i1 = new Kolicnik( new Promenljiva("x"), new Konstanta(2));
    Izraz* i2 = new Proizvod( new Konstanta(3.14), new Promenljiva("z"));
    Izraz* i3 = new Razlika( new Promenljiva("y"), i2 );
    Izraz* i4 = new Zbir( i1, i3 );
    cout << (*i4) << " = " << i4->Vrednost( o ) << endl;
}

```

```
Izraz* i5 = i4->Uprosti(o);  
cout << (*i4) << " -> " << (*i5) << endl;  
  
delete i4;  
delete i5;  
  
return 0;  
}
```

3

Enciklopedija

Saša Malkov: <http://codd.matf.bg.ac.yu/ors/files2004smalkov/enciklopedija/studenti.enciklopedija.pdf>

4

Prostor imena

Svaki objekat, funkcija, tip ili šablon koji su deklarirani u globalnoj oblasti važenja uvode neki globalni entitet. Svaki globalni entitet koji je uveden u globalnoj oblasti važenja prostora imena **mora imati jedinstven naziv**.

To znači da ukoliko želimo da koristimo neku biblioteku u svom programu moramo voditi računa da nazivi globalnih entiteta u našem programu ne dođu u koliziju sa nazivima globalnih entiteta iz biblioteke.

Kako možemo da budemo sigurni da nazivi globalnih entiteta u našem programu neće doći u sukob sa već postojećim nazivima koji su deklarirani u bibliotekama koje koristi naš program?

Ovaj problem sukobljavanja imena rešava se uz pomoć prostora imena. Autor biblioteke može da definiše prostor imena kako bi sklonio nazive u biblioteci iz globalnog prostora imena.

Primer 1

```
#include <iostream>

//using namespace std;

// Deklarisemo globalnu promenljivu
// cije je ime cout
double cout = 3.14;

// Pisemo novi prostor imena proba1
// i u okviru njega mozemo sada da definisemo
// novu promenljivu sa istim imenom
namespace proba1 {
    int cout = 5;
    float cin = 4;
    //...
};

// ovo je nastavak prostora imena
```

```
namespace proba1 {
    // Ovdje možemo da dodamo neke nove
    // definicije. Prostor imena može
    // da sadrži i druge ugnježene definicije
    // prostora imena kao i deklaracije ili
    // definicije funkcija, objekata,
    // sablona i tipova.
};

main()
{
    // Ovo cout je lokalna promenljiva
    char* cout = "cout";

    // Referisemo se na cout iz standardnog
    // prostora imena sa std::cout.
    // Operator :: naziva se operator
    // oblasti važenja.
    // Ovdje se vrši stampanje lokalne
    // promenljive cout.
    std::cout << cout << std::endl;
    // cout

    // Sa proba1::cout i proba1::cin koristimo cout
    // i cin iz prostora imena proba1
    std::cout << proba1::cout << std::endl;
    std::cout << proba1::cin << std::endl;
    // 5
    // 4

    // Sa ::cout koristimo globalnu promenljivu
    // cout (definisanu van prostora imena)
    std::cout << ::cout << std::endl;
    // 3.14

    return 0;
}
```

Ukoliko ne želimo da uključimo ceo prostor imena ali želimo da često koristimo neko posebno ime iz tog prostora tada je moguće uključiti samo dato ime iz prostora imena čime se eliminiše potreba za stalnim korišćenjem operatora oblasti važenja.

Primer 2

```
#include <iostream>
```

```
//using namespace std;

// Koristicemo samo cout i endl iz standardne biblioteke
using std::cout;
using std::endl;

main()
{
    cout << "..." << endl;

    return 0;
}
```


5

Tokovi

Ulazno izlazna funkcionalnost programa u C++-u ostvaruje se preko biblioteke **`iostream`** (biblioteka **ulaznih i izlaznih tokova**). Ova biblioteka predstavlja objektno orijentisanu hijerarhiju klasa realizovanu putem višestrukog i virtuelnog nasleđivanja. Biblioteka **`iostream`** je komponenta standardne biblioteke jezika C++.

Ulazne operacije su ugrađene u klasu **`istream`** (input stream, ulazni tok) a izlazne u klasu **`ostream`** (output stream, izlazni tok).

1. `cin` je objekat klase `istream` vezan za standardni ulaz.
2. `cout` je objekat klase `ostream` vezan za standardni izlaz.

Klasa **`iostream`** nasleđuje obe ove klase i omogućava dvosmernu komunikaciju.

Za izlaz se koristi operator prosleđivanja `<<`. To je binarni operator koji se upotrebljava u infiksnoj notaciji: levi operand je tok kome se prosleđuju podaci a desni operand je objekat koji se prosleđuje. Rezultat je referenca na izlazni tok (objekat klase `ostream`), čime je omogućeno nadovezivanje više primena ovog operatora. Npr: `cout << x << y;`

Za ulaz se koristi operator izdvajanja `>>`. To je binarni operator koji se upotrebljava u infiksnoj notaciji. Levi operand je tok iz koga se izdvajaju podaci a desni operand je objekat čiji se sadržaj izdvaja iz toka. Rezultat je referenca na ulazni tok (objekat klase `istream`), čime je omogućeno nadovezivanje više primena ovog operatora. Npr: `cin >> x >> y;`

Primer 3 *Manipulatori.*

```
#include <iostream>
```

```
// Manipulator modifikuje interno stanje  
// objekta u/i toka.  
// Manipulator se primenjuje na objekat u/i toka  
// na isti nacin kao da su u pitanju podaci.  
// Uključujemo datoteku zaglavlja iomanip  
// koja omogućava upotrebu manipulatora  
// setw
```

```
#include <iomanip>
#include <string>

using namespace std;

main()
{
    int n = 10;
    int* p = &n;
    char c = 'a';
    string s = "ovo je niska";
    char cs[] = "c-ovska niska";

    // iostream biblioteka pruza podrsku
    // za u/i operacije za ugradjene tipove
    // podataka

    cout << n << endl;
    cout << p << endl;
    cout << (long)p << endl;
    cout << c << endl;
    cout << s << endl;
    cout << cs << endl;

    /*
    cin >> n >> c >> s >> cs;

    cout << n << endl;
    cout << c << endl;
    cout << s << endl;
    cout << cs << endl;
    */

    cout << "Ovo se nece videti odmah...";

    // Ovime se data niska smesta u bafer
    // pridruzen objektu cout. Ovaj bafer
    // se prazni iz nekoliko razloga
    // (navescemo dva) tako sto se njegov sadrzaj
    // ispisuje na standardnom izlazu
    // 1. bafer se popunio da kraja pa mora da se isprazni
    // 2. eksplicitno praznjenje, npr pomocu manipulatora
    // flush, endl ili ends
```

[illegible]

5.1 Datotečni tokovi

Ulazno izlazne operacije nad datotekama obezbeđuju klase **ofstream** (izlazni datotečni tok), **ifstream** (ulazni datotečni tok) i **fstream** (datotečni tok za ulaz i izlaz).

Primer 4 *Rezultat rada programa su dve datoteke u koje je prepisan sadržaj datoteke "tokovi2.cpp".*

```
#include <iostream>

// Uključujemo biblioteku za rad sa datotekama
#include <fstream>

using namespace std;

main()
{
    // Kreira se ulazni tok inf i on se vezuje za datoteku
    // po imenu "tokovi2.cpp"
    ifstream inf("tokovi2.cpp");

    // Kreira se izlazni tok outf i on se vezuje
    // za datoteku (koja se po potrebi kreira) "izlaz.txt"
    ofstream outf("izlaz.txt");

    while(1){
        // U karakter c smesta se jedan bajt iz ulaznog
        // datotecnog toka koristeći metod get
        // koja iz toka izdvaja jedan bajt
        char c = inf.get();

        // Operator ! primenjen na objekat toka
        // vraca 1 ukoliko citanje nije uspeo
        // Primetimo da !(f) nije isto sto i f

        if( !inf )
            break;

        // U izlazni tok upisuje se karakter c koristeći
        // metod put koji u izlazni tok upisuje jedan bajt
        outf.put(c);
    }

    // Zatvara se veza izmedju toka outf i datoteke izlaz.txt
    outf.close();
}
```

```
// Izlazni tok outf vezuje se za novu datoteku izlaz2.txt
// Ova datoteka se otvara za pisanje
// prilikom cega se uklanja njen prethodni sadrzaj

outf.open("izlaz2.txt");

// ekvivalento sa
// outf.open("izlaz2.txt", ios::out);

// Drugi argument prilikom otvaranja ulazne
// ili izlazne datoteke moze biti kombinacija
// (disjunkcija na nivou bitova)
// nekih od narednih konstanti
// definisanih u klasi ios:
// ios::in otvaranje za citanje
// ios::out otvaranje za pisanje
// ios::app pri pisanju se vrsi dopisivanje
//          na postojeci sadrzaj datoteke
// ios::trunc ako datoteka postoji brise se
//          postojeci sadrzaj
// ios::ate otvaranje bez brisanja sadrzaja
//          pozicioniranje na kraj datoteke
// ios::nocreate ako datoteka ne postoji
//          otvaranje ne uspeva
// ios::noreplace ako datoteka postoji
//          otvaranje ne uspeva
// ios::binary otvaranje u binarnom rezimu
//          umesto u znakovnom

// Brise se bit ulaznog toka koji je
// bio postavljen prilikom dolaska do
// kraja fajla. Na ovaj nacin se ulazni
// tok ponovo dovodi u ispravno stanje
inf.clear( ios::eofbit );

// Ulazni tok se postavlja na pocetak
// Metod seekg pomera citanje na apsolutnu
// poziciju u toku (u ovom slucaju na nultu)
inf.seekg(0);

// Ponovo vrsimo prepisivanje datoteke, samo
// sada u drugu izlaznu datoteku
```

```
while(1){
    char c;
    inf >> c;
    if( !inf )
        break;
    outf << c;
}
// Zatvara se veza izmedju izlaznog toka i datoteke
// i izmedju ulaznog toka i datoteke
outf.close();
inf.close();
return 0;
}
```

Primer 5 *Izračunavanje dužine datoteke.*

```
#include <iostream>
#include <fstream>

using namespace std;

unsigned long velicinaDatoteke( char* s )
{
    // Ulazni tok ifs vezuje se za datoteku cije se ime nalazi
    // u nizu karaktera s
    ifstream ifs(s);

    // Metod seekg postavlja poziciju ulaznog toka na kraj. Prvi argument
    // je relativna pozicija u toku u odnosu na drugi argument koji moze biti
    // pocetak ios::beg, trenutna pozicija ios::cur ili kraj ios::end
    ifs.seekg( 0, ios::end );

    // Funkcija tellg vraca trenutnu poziciju u toku
    return ifs.tellg();
}

main()
{
    cout << "Velicina datoteke 'tokovi3.cpp' je "
          << velicinaDatoteke( "tokovi3.cpp" )
          << " bajtova."
          << endl;

    return 0;
}
```

```
}
/* Velicina datoteke 'tokovi3.cpp' je 401 bajtova. */
```

Primer 6 *Pisanje i čitanje struktura iz toka.*

```
#include <iostream>
#include <fstream>

using namespace std;

// Struktura student sadrzi podatke o studentu
struct student
{
    int indeks;
    int godina;
    char ime[50];
    char prezime[50];
};

// Funkcija pisanje upisuje niz studenata u izlaznu datoteku
void pisanje()
{
    // Niz studenata se inicijalizuje
    student studenti[] = {
        { 25, 2002, "Pera", "Peric" },
        { 31, 2001, "Zika", "Zikic" },
        { 17, 1999, "Persa", "Persic" }
    };

    // Izlazni tok f vezuje se za datoteku
    // "studenti.dat" koja se otvara u
    // binarnom rezimu rada
    ofstream f( "studenti.dat", ios::binary );

    // Prolazimo kroz niz studenata
    for( int i=0; i<sizeof(studenti)/sizeof(student); i++ )
        // Upisujemo svakog studenta u datoteku.
        // Koristimo metodu write klase ostream
        // koja omogucava da se u izlazni tok
        // stavlja odredjen broj znakova
        // ukljucujuci i završne znake ako se
        // oni nalaze u nizu.
        // Ona prima dva argumenta:
        // write(const char* s, streamsize duzina)
        // pokazivac na nisku znakova i duzinu tj
        // broj znakova za izlazni tok
```

```
        // Adresu i-tog studenta smo konvertovali
        // u pokazivac na char, a duzinu koju koristimo
        // prilikom upisivanja u tok je velicina
        // strukture student
        f.write( (char*)&(studenti[i]), sizeof(student));

        f.close();

}

void citanje( int i )
{
    student s;

    // Ulazni tok vezujemo za datoteku otvorenu
    // u binarnom rezimu rada
    ifstream f( "studenti.dat", ios::binary );

    // U ulaznoj datoteci postavljamo se na
    // odgovarajuće mesto
    f.seekg( i * sizeof(student));

    // Metod read klase istream ima sledeci potpis
    // read( char* adresa, streamsize size)
    // i funkcioniše inverzno u odnosu na funkciju
    // write: ona izdvaja size susednih bajtova iz
    // ulaznog toka i smesta ih u memoriji sa
    // pocetkom na adresi adresa
    f.read( (char*)&s, sizeof(student));

    // Proverava se da li je citanje uspelo
    if( !f )
    // ukoliko nije stampa se odgovarajuća poruka
    cout << "Ne postoji " << i << ". student." << endl;
    // koliko jeste stampaju se podaci o studentu
    else
    {
        cout << s.indeks << ' '
              << s.godina << ' '
              << s.ime << ' '
              << s.prezime << endl;
    }
}
```



```
        f.close();
    }

main()
{
    pisanje();
    citanje(2);
    citanje(1);
    citanje(7);
    return 0;
}

/*
17 1999 Persa Persic
31 2001 Zika Zikic
Ne postoji 7. student.
*/
```


6

Izuzeci

Izuzeci su anomalije koje program otkriva u vreme svog izvršavanja, kao što su na primer deljenje nulom, pristupanje nizu van njegovih granica i slično. Obzirom da ovakvi izuzeci ne predstavljaju deo normalnog funkcionisanja programa, očekuje se da program trenutno reaguje na njih. Upravljanje izuzecima je mehanizam koji omogućava komunikaciju, odnosno prenošenje izuzetka između dva nepovezana (često nezavisno razvijana) dela programa. Jezik C++ poseduje već gotove alatke za proglašavanje izuzetaka kao i ispravno upravljanje njima.

Naime, kada se izuzetak uoči, deo programa koji ga je otkrio može da proglašavanjem ili ispaljivanjem (engl. *throwing*) izuzetka saopšti da se taj izuzetak pojavio. Izuzetak se ispaljuje pomoću izraza za ispaljivanje koji se sastoji od ključne reči **throw** iza koje zatim dolazi izraz čiji tip odgovara tipu ispaljenog izuzetka. Sve naredbe koje mogu da ispale izuzetak moraju se nalaziti u okviru bloka `try`. Ovaj blok započinje ključnom rečju **try** iza koje, unutar vitičastih zagrada, sledi niz programskih naredbi. Iza bloka `try` sledi lista hvatača koji se nazivaju klauzule za hvatanje (engl. *catch clause*). Blok `try` povezuje skup naredbi koje mogu da ispale izuzetak sa skupom hvatača koji na odgovarajući način obrađuju te izuzetke. Klauzula za hvatanje se sastoji iz tri dela: ključne reči **catch**, deklaracije jednog tipa ili jednog objekta u zagradama (to se zove još i deklaracija izuzetka) i skupa naredbi u okviru jedne složene naredbe.

Kada dođe do izuzetka, sve naredbe koje dolaze iza naredbe koja je proizvela izuzetak se preskaču a program nastavlja izvršenje u klauzuli za hvatanje koja upravlja tim izuzetkom. Tada se izvršava složena naredba klauzule za hvatanje koja je izabrana a zatim se izvršavanje programa nastavlja u naredbi koja sledi iza poslednje klauzule za hvatanje u listi. Ukoliko ne postoji odgovarajuća klauzula za hvatanje koja je u stanju da rukuje izuzetkom, izvršavanje se nastavlja u funkciji `terminate()` koja je definisana u standardnoj biblioteci jezika C++.

Blok `try` uvodi lokalnu oblast važenja tako da se promenljivim deklarisanim u bloku `try` ne može pristupati izvan tog bloka pa čak ni iz klauzula za hvatanje.

Postoji i hvatač (klauzula za hvatanje) koja obrađuje sve izuzetke i ona ima deklaraciju izuzetka u formi (...). U ovu klauzulu za hvatanje se ulazi za bilo koji tip izuzetka. Klauzule za hvatanje se uvek ispituju jedna po jedna onim redosledom kojim su navedene iza bloka `try`. Kada odgovarajuća klauzula bude pronađena ostale

se ne ispituju. To znači da ako se klauzula `catch(...)` koristi u kombinaciji sa drugim klauzulama za hvatanje, ona se mora postaviti da bude poslednja u listi hvatača, inače će kompajler prijaviti grešku u fazi prevođenja.

Primer 7 *Primer koji ilustruje jednostavno upravljanje izuzecima.*

```
#include <iostream>
using namespace std;

main()
{
    cout << "pre bloka try" << endl;
    try {
        cout << "pre throw" << endl;
        throw "namerni izuzetak";
        cout << "posle throw" << endl;
    }
    catch( char* s ){
        cout << "Doslo je do greske: " << s << endl;
    }
    catch(...){
        cout << "Nepoznata greska!" << endl;
    }
    cout << "posle bloka try" << endl;
    return 1;
}

/* Izlaz:
pre bloka try
pre throw
Doslo je do greske: namerni izuzetak
posle bloka try
*/
```

Primer 8 *Primer koji ilustruje funkciju koja proizvodi izuzetke i program koji njima upravlja.*

```
#include <iostream>
using namespace std;

int f( int x )
{
    if( x<0 )
        throw "Parametar funkcije f je manji od 0!";
    if( x>10)
```

```

        throw "Parametar funkcije f je veci od 10!";
    return x * x;
}

main()
{
    cout << "pocetak" << endl;
    for( int i=-1; i<=12; i++ ){
        try {
            int y = f(i);
            cout << "f(" << i << ") = " << y << endl;
        }
        catch( char* s ){
            cerr << "GRESKA: " << s << endl;
        }
    }
    cout << "kraj" << endl;
    return 1;
}

```

```

/* Izlaz:
pocetak
GRESKA: Parametar funkcije f je manji od 0!
f(0) = 0
f(1) = 1
f(2) = 4
f(3) = 9
f(4) = 16
f(5) = 25
f(6) = 36
f(7) = 49
f(8) = 64
f(9) = 81
f(10) = 100
GRESKA: Parametar funkcije f je veci od 10!
GRESKA: Parametar funkcije f je veci od 10!
kraj
*/

```

Primer 9 *Primer koji ilustruje korišćenje klauzule za hvatanje koja obrađuje sve izuzetke.*

```

#include <iostream>
using namespace std;

int f( int x )

```

```
{
    if( x<0 )
        throw "Parametar funkcije f je manji od 0!";
    if( x>10)
        throw x;
    return x * x;
}

void g( int a, int b)
{
    for( int i=a; i<=b; i++ ){
        try {
            int y = f(i);
            cout << "f(" << i << ") = " << y << endl;
        }catch( char* s ){
            cerr << "GRESKA: " << s << endl;
        }
    }
}

main()
{
    cout << "pocetak" << endl;
    try {
        g( -1, 12 );
    }catch(...){
        cerr << "Nepoznata greska!" << endl;
    }
    cout << "kraj" << endl;
    return 1;
}

/* Izlaz:
pocetak
GRESKA: Parametar funkcije f je manji od 0!
f(0) = 0
f(1) = 1
f(2) = 4
f(3) = 9
f(4) = 16
f(5) = 25
f(6) = 36
f(7) = 49
f(8) = 64
```

```
f(9) = 81
f(10) = 100
Nepoznata greska!
kraj
*/
```

Primer 10 *Primer koji ilustruje kako se nakon ispaljivanja izuzetka u okviru neke funkcije ta funkcija napusta i pozivaju se podrazumevani destruktori objekata deklarisanih u okviru te funkcije.*

```
#include <iostream>
using namespace std;

class A
{
public:
    A( int a )
        : _a(a)
    { cerr << "A::A(" << _a << ")" << endl; }
    ~A()
        { cerr << "A::~A(" << _a << ")" << endl; }

private:
    int _a;
};

int f( int x )
{
    A a(x);
    // Ukoliko dodje do ispaljivanja izuzetka,
    // f-ja f() se napusta i vrsi se implicitno pozivanje
    // destruktora klase A radi unistavanja objekta a.
    if( x<0 )
        throw "Parametar funkcije f je manji od 0!";
    if( x>3)
        throw x;
    return x * x;
}

void g( int a, int b)
{
    for( int i=a; i<=b; i++ ){
        try {
            int y = f(i);
            cout << "f(" << i << ") = " << y << endl;
        }catch( char* s ){
```

```

        cerr << "GRESKA: " << s << endl;
    }
}

main()
{
    cout << "pocetak" << endl;
    try {
        g( -1, 5 );
    }catch(...){
        cerr << "Nepoznata greska!" << endl;
    }
    cout << "kraj" << endl;
    return 1;
}

/* Izlaz:
pocetak
A::A(-1)
A::~~A(-1)
GRESKA: Parametar funkcije f je manji od 0!
A::A(0)
A::~~A(0)
f(0) = 0
A::A(1)
A::~~A(1)
f(1) = 1
A::A(2)
A::~~A(2)
f(2) = 4
A::A(3)
A::~~A(3)
f(3) = 9
A::A(4)
A::~~A(4)
Nepoznata greska!
kraj
*/

```

Primer 11 *Primer koji ilustruje kako se može desiti da usled ispaljivanja izuzetka neki resursi ostanu zauvek neoslobodeni.*

```

#include <iostream>
using namespace std;

```



```
class A
{
public:
    A( int a )
        : _a(a)
        { cerr << "A::A(" << _a << ")" << endl; }
    ~A()
        { cerr << "A::~A(" << _a << ")" << endl; }

private:
    int _a;
};

int f( int x )
{
    A a(x);
    //Ukoliko funkcija moze da dobije odredjeni resurs
    //(otvori datoteku ili da alocira memoriju na hipu)...
    A* p = new A(100+x);
    //...i ukoliko dodje do ispaljivanja izuzetka, izlazi se iz funkcije...
    if( x<0 )
        throw "Parametar funkcije f je manji od 0!";
    if( x>3)
        throw x;
    //...i oslobadjanje ovog resursa se nece obaviti.
    delete p;
    return x * x;
}

void g( int a, int b)
{
    for( int i=a; i<=b; i++ ){
        try {
            int y = f(i);
            cout << "f(" << i << ") = " << y << endl;
        }catch( char* s ){
            cerr << "GRESKA: " << s << endl;
        }
    }
}

main()
{
    cout << "pocetak" << endl;
```

```

    try {
        g( -1, 5 );
    }catch(...){
        cerr << "Nepoznata greska!" << endl;
    }
    cout << "kraj" << endl;
    return 1;
}

```

```

/* Izlaz:
pocetak
A::A(-1)
A::A(99)
A::~~A(-1)
GRESKA: Parametar funkcije f je manji od 0!
A::A(0)
A::A(100)
A::~~A(100)
A::~~A(0)
f(0) = 0
A::A(1)
A::A(101)
A::~~A(101)
A::~~A(1)
f(1) = 1
A::A(2)
A::A(102)
A::~~A(102)
A::~~A(2)
f(2) = 4
A::A(3)
A::A(103)
A::~~A(103)
A::~~A(3)
f(3) = 9
A::A(4)
A::A(104)
A::~~A(4)
Nepoznata greska!
kraj
*/

```

Primer 12 *Primer koji ilustruje kako se korišćenjem hvatača koji obrađuje sve izuzetke može sprečiti da usled ispaljivanja izuzetka neki resursi ostanu zauvek neoslobodeni.*

```
#include <iostream>
```

```
using namespace std;

class A
{
public:
    A( int a )
        : _a(a)
        { cerr << "A::A(" << _a << ")" << endl; }
    ~A()
        { cerr << "A::~A(" << _a << ")" << endl; }

private:
    int _a;
};

int f( int x )
{
    A a(x);
    A* p = new A(100+x);
    try {
        if( x<0 )
            throw "Parametar funkcije f je manji od 0!";
        if( x>3)
            throw x;
        cout << "Poziv destruktora u bloku try" << endl;
        delete p;
    }
    catch(...){
        cout << "Poziv destruktora u hvatacu izuzetaka" << endl;
        // Obezbedjuje da se memorija na koju pokazuje p oslobodi
        // i u slucaju ispaljivanja izuzetka.
        delete p;
        //Ispaljuje izuzetak dalje prosledjujuci ga drugoj
        //klauzuli za hvatanje radi obradjivanja.
        throw;
    }

    return x * x;
}

void g( int a, int b)
{
    for( int i=a; i<=b; i++ ){
        try {
```

```

        int y = f(i);
        cout << "f(" << i << ") = " << y << endl;
    }catch( char* s ){
        cerr << "GRESKA: " << s << endl;
    }
}

main()
{
    cout << "pocetak" << endl;
    try {
        g( -1, 5 );
    }catch(...){
        cerr << "Nepoznata greska!" << endl;
    }
    cout << "kraj" << endl;
    return 1;
}

```

```

/* Izlaz:
pocetak
A::A(-1)
A::A(99)
Poziv destruktora u hvatacu izuzetaka
A::~~A(99)
A::~~A(-1)
GRESKA: Parametar funkcije f je manji od 0!
A::A(0)
A::A(100)
Poziv destruktora u bloku try
A::~~A(100)
A::~~A(0)
f(0) = 0
A::A(1)
A::A(101)
Poziv destruktora u bloku try
A::~~A(101)
A::~~A(1)
f(1) = 1
A::A(2)
A::A(102)
Poziv destruktora u bloku try
A::~~A(102)

```

```
A::~~A(2)
f(2) = 4
A::A(3)
A::A(103)
Poziv destruktora u bloku try
A::~~A(103)
A::~~A(3)
f(3) = 9
A::A(4)
A::A(104)
Poziv destruktora u hvatacu izuzetaka
A::~~A(104)
A::~~A(4)
Nepoznata greska!
kraj
*/
```


7

Zadaci sa praktikuma

7.1 Matematički izrazi

Zadatak 1 *Dopuniti klase za rad sa aritmetičkim izrazima sledećim klasama:*

- *Napisati klasu `UnarniOperator` koji predstavlja apstrakciju svih unarnih operacija i funkcija.*
- *Napisati klasu `UnarniMinus` koja predstavlja unarnu negaciju.*
- *Napisati klase `FnSin` i `FnCos` koje predstavljaju analitičke funkcije \sin i \cos .*
- *Napisati klase `FnLn` i `FnExp` koje predstavljaju analitičke funkcije \ln i e^x .*
- *Obezbediti računanje izvoda proizvoljnog izraza po proizvoljnoj promenljivoj.*

//Resenje: Branislav Zelenak

```
class UnarniOperator : public Izraz
{
public:
    UnarniOperator( Izraz* i )
        : _I(i)
    {}
    ~UnarniOperator()
    {
        delete _I;
    }
    UnarniOperator( const UnarniOperator& z )
        : _I( z._I->Kopija() )
    {}
    UnarniOperator& operator = ( const UnarniOperator& z )
    {
        if( this != &z ){
            delete _I;
```

```

        _I = z._I->Kopija();
    }
    return *this;
}

double Vrednost( const Okolina& o ) const
{ return Izracunaj( _I->Vrednost(o)); }

Izraz* Uprosti( const Okolina& o ) const
{
    Izraz* i = _I->Uprosti(o);
    if( i->JesteKonstanta()){
        Izraz* r = new Konstanta( Izracunaj( i->Vrednost(o)));
        delete i;
        return r;
    }
    else
        return NapraviNovi(i);
}

void Ispisi( ostream& ostr ) const
{ ostr << "( " << Naziv() << ' ' << *_I << ')'; }

protected:
    virtual double Izracunaj( double x) const = 0;
    virtual Izraz* NapraviNovi( Izraz* i ) const = 0;
    virtual string Naziv() const = 0;

private:
    Izraz *_I;
};

class UnarniMinus : public UnarniOperator
{
public:
    UnarniMinus( Izraz* i)
    :UnarniOperator( i )
    {}
    UnarniMinus* Kopija() const
    { return new UnarniMinus(*this); }
protected:
    double Izracunaj( double x) const
    { return -x;}
    Izraz* NapraviNovi( Izraz* i) const
    { return new UnarniMinus( i ); }

```



```
        string Naziv() const
            { return "UNARNI MINUS"; }
};

class FnSin : public UnarniOperator
{
public:
    FnSin( Izraz* i)
        :UnarniOperator( i )
    {}
    FnSin* Kopija() const
        { return new FnSin(*this); }
protected:
    double Izracunaj( double x) const
        { return sin(x);}
    Izraz* NapraviNovi( Izraz* i) const
        { return new FnSin( i ); }
    string Naziv() const
        { return "FNSIN"; }
};

class FnCos : public UnarniOperator
{
public:
    FnCos( Izraz* i)
        :UnarniOperator( i )
    {}
    FnCos* Kopija() const
        { return new FnCos(*this); }
protected:
    double Izracunaj( double x) const
        { return cos(x);}
    Izraz* NapraviNovi( Izraz* i) const
        { return new FnCos( i ); }
    string Naziv() const
        { return "FNCOS"; }
};

class FnLn : public UnarniOperator
{
public:
    FnLn( Izraz* i)
        :UnarniOperator( i )
    {}
};
```

```

    FnLn* Kopija() const
        { return new FnLn(*this); }
protected:
    double Izracunaj( double x) const
        { return log(x);}
    Izraz* NapraviNovi( Izraz* i) const
        { return new FnLn( i ); }
    string Naziv() const
        { return "FNLN"; }
};

class FnExp : public UnarniOperator
{
public:
    FnExp( Izraz* i)
        :UnarniOperator( i )
        {}
    FnExp* Kopija() const
        { return new FnExp(*this); }
protected:
    double Izracunaj( double x) const
        { return exp(x);}
    Izraz* NapraviNovi( Izraz* i) const
        { return new FnExp( i ); }
    string Naziv() const
        { return "FNEXP"; }
};

```

7.2 Enciklopedija

Zadatak 2 *Izmeniti internu strukturu hijerarhije klasa za rad sa enciklopedijskim podacima.*

7.3 Tokovi i izuzeci

Zadatak 3 *Napisati program koji broji znakove, reči i linije u datoteci čije se ime zadaje kao argument komandne linije.*

Zadatak 4 *Napisati program koji čita podatke iz datoteke `ulaz.txt` i na osnovu učitane vrednosti iz datoteke računa vrednosti funkcija `arcsin`, `arccos`, `1/x`, `log(a,b)` (logaritam od a u osnovi b), `koren(x)` ili ispisuje poruku o greci u izlaznu datoteku `izlaz.txt`. Zadatak rešavati upotrebom izuzetaka.*

Na primer, ako datoteka `ulaz.txt` sadrži vrednosti
0.5 0.7 0.8 10 -10 1 2 3 -3 - 4

tada datoteka izlaz.txt treba da sarži sledeći tekst:

```
arccos(0.5) = 1.0472
arcsin(0.5) = 0.523599
koren(0.5) = 0.707107
JedanKrozIks(0.5) = 2
log(0.5,0.7) = 0.514573
arccos(0.8) = 0.643501
arcsin(0.8) = 0.927295
koren(0.8) = 0.894427
JedanKrozIks(0.8) = 1.25
log(0.8,10) = -10.3189
Greska za x = -10: arccos: argument nije u intervalu [-1 1].
Greska za x = -10: arcsin: argument nije u intervalu [-1 1].
Greska za x = -10: koren: potkorena velicina mora biti nenegativna!!!
JedanKrozIks(-10) = -0.1
Greska: log: osnova mora biti pozitivna i razlicita od jedan.
Greska za x = 2: arccos: argument nije u intervalu [-1 1].
Greska za x = 2: arcsin: argument nije u intervalu [-1 1].
koren(2) = 1.41421
JedanKrozIks(2) = 0.5
log(2,3) = 1.58496
Greska za x = -3: arccos: argument nije u intervalu [-1 1].
Greska za x = -3: arcsin: argument nije u intervalu [-1 1].
Greska za x = -3: koren: potkorena velicina mora biti nenegativna!!!
JedanKrozIks(-3) = -0.333333
Greska: Nema dovoljno argumenata u datoteci za log!
```